File/Data Handling

Reading/writing Data from/to Self−described Formats

A variable (defined in more detail in Reading, creating, and altering variables) can be obtained from a file or collection of files, or can be generated as the result of a computation. Files can be in any of the self−describing formats netCDF, HDF, GrADS/GRIB (GRIB with a GrADS control file) or xml files generated by CDMS.

For instance, to read the variable u from file sample.nc:

```
import cdms
f = cdms.open('sample.nc')
u = f('u')
```

Data can be read by index or by world coordinate values. The following reads the n−th timepoint of u:

```
u0 = f('u',time=slice(n,n+1))
```

and this reads u at time 366.0:

```
u1 = f('u',time=366.)
```

If an OPeNDAP (formerly known as DODS) enabled version of CDAT was installed, datasets on remote OPeNDAP servers can be accessed:

```
f = cdms.open('http://cdc.noaa.gov/cgi-bin/nph-nc/Datasets/ncep/air.1990.nc')
```

A variable can be written to a file with the write function:

```
g = cdms.open('sample2.nc','w')
g.write(u)
# <Variable: u, file: sample2.nc, shape: (1, 16, 32)>
g.close()
```

More details on reading and writing data can be found in Climate Data Management System (cdms.pdf).

Reading/writing Fortran Formatted Data

The Scientific Python package maintained by Konrad Hinsen contains numerous subpackages useful in scientific applications. The "IO" subpackage is useful for reading and writing data using Fortran format statements. The example shown below demonstrates how easy it is to read Fortran formatted data.

```
f = open(ascii_filename, 'r')

# Import the module that does the work.
from Scientific.IO import FortranFormat

# Declare the fortran formats used to create the data.
ff1 = FortranFormat.FortranFormat('2i6')
ff2 = FortranFormat.FortranFormat('12i6')

data_line = f.readline()
mon, yr = FortranFormat.FortranLine(data_line, ff1)
```

```
# Now define an array to read the data into.
import Numeric
T_array = Numeric.zeros((14,))

# Assign the values.
T_array[start_index: end_index] = FortranFormat.FortranLine(f.readline(), ff2)

# Note: You must have previously defined T_array.
# See tutorial examples for more details.
```

Reading Data From ASCII Text Files (asciidata)

The asciidata module is useful in reading (and writing) data that is in ASCII format with the ability to specify tab or comma or space delimited fields. This is particularly useful to deal with importing (or exporting) spreadsheet data. For example:

```
import asciidata
time, pressure = asciidata.comma_delimited('myfile.txt')
```

The IO subpackage of Scientific also contains other useful facilities of this type.

Reading/writing Unformatted Data (binaryio)

To handle binary or unformatted data, the module binaryio offers a convenient interface. The following example illustrates the use of this module. Note that binary files are platform and compiler dependent.

```
# To write binary data.
>>> import binaryio
>>> iunit = binaryio.bincreate('filename')
>>> binaryio.binwrite(iunit, my_array_with_upto_4_dimensions)
>>> binaryio.binclose(iunit)

# To read binary data
>>> iunit = binaryio.binopen('filename')
>>> y = binaryio.binread(iunit, n, ...)
>>> binaryio.binclose(iunit)
```

Reading, Creating, and Altering Variables

Climate data comes in many different file formats, organized in many different ways. The cdms package gives you a uniform interface so that you can write processing algorithms and graphics that will work with a wide variety of different data formats.

In CDMS, the basic unit of data is the variable. A variable is essentially a multidimensional array, augmented with a domain and with metadata. The domain describes the spatial location and temporal information associated with the array. For example, a variable on an orthogonal grid may have a domain consisting of (but not limited to) time, level, latitude, and longitude axes. The metadata associated with a variable consists of a collection of attribute−value pairs. The set of attributes of a variable is not predefined, although some attributes, such as the units and the missing data value, are commonly used.

A variable may be stored in a single physical file or in a collection of files, called a dataset. The files themselves may be in any of several self−describing formats, such as netCDF, HDF, and GraDS/GRIB. CDMS provides a uniform interface to data, so that the same processing algorithm or graphics routine will work with any of the supported data formats.

For computing tasks, variables can be used much like arrays. The common arithmetic functions are defined for variables, as well as I/O and slicing operations. One advantage of using CDMS variables for computation is that the associated domain and metadata information is carried along with the computation. The benefit of this approach is that, for example, if we average over time, and then plot the result, the plotting routines can still be aware that the other dimensions represent latitude and longitude and draw the continental outlines, do projections correctly, etc. If the result were merely the averaged data, that interpretation of it would have been lost. This frequently results in much simpler scripts than otherwise would be the case.

However, CDAT is also designed for extensibility. The capability to add external C or Fortran modules is extremely important for advanced applications. For example, many external modules use the "array" type defined by the Numerical Python (NumPy) module. To accomodate the need for interoperability, CDAT provides a hierarchy of representations for data arrays:

- Numeric Array: a multidimensional array, all elements have the same datatype (real, integer etc.). This type is supported by the Numerical Python (NumPy) module.
- Masked Array (MA): A Numeric array with an optional missing data mask. Operations on these compute the mask of the result.
- Masked Variable (MV): A masked array having a domain and metadata. Computations carry along the domain and metadata information where possible. A masked variable in memory is referred to as transient variable and a masked variable in a dataset is called a file variable.

A schematic of the array hierarchy is shown in the figure below.

While the above figure represents the general schematic of the different array constructs, the user should be aware that the individual constructs also include functions (for e.g. arithmetic functions) that operate on these arrays. Some of these functions are intended to convert arrays of one type to the other. These allow the user to convert the array to the necessary level to suit the algorithm. The conversion from one level of array abstraction to the other is dealt with in the following sections.

Constructing Numeric Arrays

As explained in the Numerical Python manual, a Numeric array can be constructed in many ways. The basic usage is to apply the array "constructor" Numeric.array:

```
import Numeric
x = Numeric.array (s)
```

where s can be a Python list, tuple, or another Numeric array.

For many applications, space allocation for newly defined variables needs to be controlled so as not to run out of memory. CDAT allows for some control over this. For example, ff you do not require a separate copy of the data, copy=0 can be added.

```
x = Numeric.array(s, copy=0)
```

If you want to control the type of the data, you can supply a typecode, usually in the form of one of the abstract constants supplied in Numeric for this purpose. For example, the following would create y as an array containing the floating−point numbers 1., 2., and 3.:

```
y = Numeric.array([1,2,3], Numeric.Float)
```

By the way, a very frequent beginner error is to say something like:

```
y = Numeric.array(1,2,3)  # Error !
```

which is an error that will result in a very strange message:

```
import Numeric
y = Numeric.array(1,2,3)
Traceback (most recent call last):
File "<pyshell#1>", line 1, in ?
y = Numeric.array(1,2,3)
TypeError: typecode argument must be a string.
```

What has happened is that the second argument to Numeric.array was the number 2, and Numeric is expecting one of its typecode letters such as 'd' in that position.

Numeric to MA

Given a Numeric array x, if we wish to construct a masked array, it will be one of these cases:

- We want to treat x as a masked array xm, but none of its values are currently missing, and wish to share x's data space, so that a modification to xm is also a modification to x:

```
import M
xm = MA.masked_array (x)
```

- We want to treat x as a masked array xm with no values considered missing and without sharing x's space:

```
xm = MA.array (x)
```

- We want to treat x as a masked array xm with a certain value v treated as a missing value. See the MA manual for other arguments, such as controlling the precision with which a value in x must be equal to v in order to be considered missing.

```
xm = MA.masked_value (x, v).
```

- We want to treat x as a masked array xm but with those values considered missing that correspond to non−zero values in an array m of zeros and ones. Again, see the MA manual for more options:

```
xm = MA.array(x, mask=m).
```

- The array x is of a numeric type and we want to mask all those values greater than a certain value v:

```
xm = MA.masked_greater (x, v).
```

Similarly, there are functions masked_greater_equal, masked_less, masked_less_equal, masked_equal and masked_where.

Numeric or MA to Transient Variable (MV)

The array constructor cdms.MV.array is similar to the MA.array constructor but allows additional arguments specifying the metadata. For full usage, see Climate Data Management System (cdms.pdf). The options discussed in the previous section will produce transient variables as long as you use the functions in cdms.MV instead of those in MA.

The most frequent additional argument to MV.array is to specify a list of axes using the axes=alist argument. Often these axes have been extracted from another variable using getAxis or getAxisList, or created from data.

For example, if you were going to operate on variable v using some process that returns a Numeric array, and in the process remove its time dimension, you might do something like this:

```
import cdms, MV
f = cdms.open('../data_directory/clt.nc')
v = f('clt')
alist = v.getAxisList(omit='time')
u = v(time=('1979-1-1', '1979-1-1'), squeeze=1)
x = MV.array(u, axes=alist)
```

Now, x is again a transient variable and it has the geographic information reattached so that plots will show the continents. Note that it is usually not necessary to do this. Any ordinary arithmetic, and any use of the functions in MV, will return transient variables with appropriate axes.

As mentioned above, transient variables can have associated attributes which are accessed and set using the Python dot notation:

```
u.units='m/s'
print u.units
m/s
```

Attribute values can be strings, scalars, or 1−D Numeric arrays. More details on variables and the available functions and methods can be found in Climate Data Management System (cdms.pdf).

MA or Transient Variable to Numeric

Many packages that support Numerical Python arrays have been developed. You can find links to some of them on the CDAT website. These packages work with Numerical Python arrays. For these and other applications (such as modules that are written in Fortran that expect data as Numeric arrays), it is necessary to convert existing arrays that are Transient Variables or Masked Arrays to Numeric arrays. This is easily accomplished using the filled method. For example:

```
import cdms, MV
f = cdms.open('../data_directory/clt.nc')
v = f('clt')

# If we want the Numeric array to have 1.e+20
# where the value is missing.
v_array = MV.filled(v, 1.e+20)
```

The figure below summarizes the functions we have just seen.

- **Handling Time**

The cdtime module implements the cdms time types, methods, and calendars. These are made available with the command:

```
import cdtime
```

Two time types are available: relative time and component time. Relative time is time relative to a fixed base time. It consists of:

- ♦ a units string, of the form "units since basetime", and
- ♦ a floating−point value

For example, the time "28.0 days since 1996−1−1" has value=28.0, and units="days since 1996−1−1" Component time consists of the integer fields year, month, day, hour, minute, and the floating−point field second. A sample component time is 1996−2−28 12:10:30.0

The cdtime module contains functions for converting between these forms, based on the common calendars used in climate simulation.

Basic arithmetic and comparison operators are also available. Some examples are shown below.

```
# Example: Creating component and relative times.
import cdtime
c = cdtime.comptime(1996,2,28)
r = cdtime.reltime(28,"days since 1996-1-1")

# Example: Adding and subtracting times.
print r.add(1,Days)
29.00 days since 1996-1-1

print c.add(36,Hour)
1996-2-29 12:0:0.0

print r.sub(10,Days)
18.00 days since 1996-1-1

print c.sub(30,Days)
1996-1-29 0:0:0.0

# Example: Comparing times.
r = cdtime.reltime(28,"days since 1996-1-1")
c = cdtime.comptime(1996,2,28)
print r.cmp(c)
-1

print c.cmp(r)
1

print r.cmp(r)
0
```

```
# Example: Extracting year/month/day information
print c.year
1996

print c.month
2

# Example: Converting between time types.
print r.tocomp()
1996-1-29 0:0:0.0

print c.torel("days since 1996-1-1")
58.00 days since 1996-1-1

print r.torel("days since 1995")
393.00 days since 1995

print r.torel("days since 1995").value
393.0
```

More details of the cdtime module can be found in Climate Data Management System (cdms.pdf).

Axes and Domains

The spatial and temporal information associated with a variable is represented by the variable domain, an ordered tuple of axes and/or grids. In the above example, the domain of the variable u is the tuple (time, latitude, longitude). This indicates the order of the dimensions, with the slowest−varying dimension listed first (time). Each element of the tuple is an axis. An axis is like a 1−D variable, in that it can be sliced, and has attributes. A number of functions are available to access axis information. For example, to see the list of time values associated with u:

```
t = u.getTime()
print t[:]
[ 0., 366., 731.,]

# Similar methods of extracting the latitude, longitude
# and level axes or the lat-lon Grid are available
myGrid = u.getGrid()
lat_axis = u.getLatitude()

# Individual axes can also be accessed by their index
first_axis = u.getAxis(0)
```

Similarly, creating axes and domains are easily accomplished using the constructor functions. Some of the basic constructors and methods are illustrated in the tutorials. More details of dealing with axes and domains can be found in Climate Data Management System (cdms.pdf).

Data Selection

As seen previously, the cdms module is used to open files and extract data. Currently, cdms only handles data on regular (rectangular) grids. In the near future cdms will be able to handle data on irregular grids with all the same functionality. The cdms module also allows for easy selection of subsets of the data stored in files or in memory. The use of keywords to describe specific axes, "Selectors" to describe specific portions of interest, and control over the precision of extracted areas are some of the features that make this a powerful package. A few simple examples are shown here.

```
import cdms
f = cdms.open('file_name')

# To extract data for specified times
ta_1996_only =f('ta',time=('1996-1-1','1996-12-1'))

# To extract data for specified latitude and longitude areas
x =f('t', latitude=(-5.,5.), longitude=(210., 270.))

# To extract data at a single level (or any other dimension)
x = f('t', level=(250.,250.))

# The above will extract data at the prescribed level
# if that level is present in the data. It will also
# result in the level dimension being retained as a
# singleton dimension. To eliminate singleton dimensions
# set the 'squeeze' option, e.g.
x = f('t', level=(250.,250.), squeeze=1)

# Using cdutil to specify regions precisely.
import cdutil
NINO3 = cdutil.region.domain(latitude=(-5.,5.), longitude=(210., 270.))
nino3_area_exact = f('t', NINO3)

# In the above case, the precise region is returned with
# the weights and grid cell bounds set to match the
# request.
```

Some commonly used domains have been pre−defined for convenience. They are:

```
NH | NorthernHemisphere
SH | SouthernHemisphere
Tropics                    # latitude extends from -23.4 to 23.4
NPZ | AZ | ArcticZone      # latitude extends from 66.6N to 90.0N
SPZ | AAZ | AntarcticZone  # latitude extends from 90.0S to 66.6S
```

Example:

```
from cdutil import region
t_northern_hemisphere_only = f('t', region.NH)
```

The tutorial examples illustrate these features in more detail. For details of the available commands and their usage refer to Climate Data Management System (cdms.pdf).

Regridding Data

CDMS has functions to interpolate gridded data:

- from one horizontal (lat/lon) grid to another
- from one set of pressure levels to another
- from one vertical (lat/level) cross−section to another vertical cross−section.

The simplest method to regrid a variable from one horizontal, lat/lon grid to another is to use the regrid function defined for variables. This function takes the target grid as an argument, and returns the variable regridded to the target grid:

```
import cdms
f = cdms.open('../data_directory/ccc/perturb.xml')

# Read the data and check the shape of the data
rlsf = f('rls')
rlsf.shape
(4, 48, 96)

# Now choose a file that contains data in the
# desired output (target) grid.
g = cdms.open('../data_directory/mri/perturb.xml')

# Get the file variable. The data is not actually read
# in when we use square bracket pairs like so:
rlsg = g['rls']

# Get the target grid
outgrid = rlsg.getGrid()

# Apply the regrid method to get the desired result.
rlsnew = rlsf.regrid(outgrid)
rlsnew.shape
(4, 46, 72)
outgrid.shape
(46, 72)
```

A somewhat more efficient method is to create a regridder function. This has the advantage that the mapping is created only once and can be used for multiple arrays.The steps in this process are:

- Given an input grid and output grid, generate a regridder function.
- Call the regridder function on an array, resulting in an array defined on the output grid. The regridder function can be called with any array or variable defined on the input grid.

The following example illustrates this process:

```
import cdms
from regrid import Regridder

f = cdms.open('../data_directory/ccc/perturb.xml')
rlsf = f['rls']
ingrid = rlsf.getGrid()
g = cdms.open('../data_directory/mri/perturb.xml')
outgrid = g['rls'].getGrid()
regridfunc = Regridder(ingrid, outgrid)
rlsnew = regridfunc(rlsf)
f.close()
g.close()
```

For more details on regridding functions and methods please refer to Climate Data Management System (cdms.pdf).

Working with Masks

Masks were introduced earlier in Reading, creating, and altering variables. They are a convenient way to deal with data that either has missing values or where one would have to deal with masking out regions. The masks can be handled seperately from the data to keep the computational expense down and one can also use the logical and/or operators to perform complex tasks easily. A small example is shown below. More details of how to use masks are in the Numeric manual and examples of masks and masking operations in action can be found in the geting started tutorials listed in Chapter 2.

```
# Let us open a data file that contains surface type
# (land fraction) data and extract data
import cdms, MV
f_surface = cdms.open('sftlf_ta.nc')
surf = f_surface('sftlf')

# Designate land where "surf" has values
# not equal to 100
land_only = MV.masked_not_equal(surf, 100.)
land_mask = MV.getmask(land_only)

# Now extract a variable from another file
f = cdms.open('ta_1994-1998.nc')
ta = f('ta')

# Apply this mask to retain only land values.
ta_land = cdms.createVariable(ta, mask=land_mask, copy=0, id='ta_land')
```

Databases

A Database is a collection of datasets and other CDMS objects. It consists of a hierarchical collection of objects, with the database being at the root, or top of the hierarchy. A database is used to:

- search for metadata
- access data
- provide authentication and access control for data and metadata

Details of creating, altering, and searching through databases are beyond the scope of this beginners document. The interested reader should refer to Climate Data Management System (cdms.pdf).